Practical Work on the Project session 2
Finite State Machines

This work session is dedicated to introduce the use of *Finite State Machines* in *HMI*, as a way to describe either an interaction, or the different states of presentation of a component. **Duration with course: 2h**

After having read the course dedicated to *Finite State Machines* you are now going to use them to manage interactions, and to manage the visual states of a graphical component for your project.

Exercise 1: *FSM* to manage interactions (*pan* & *drag*)

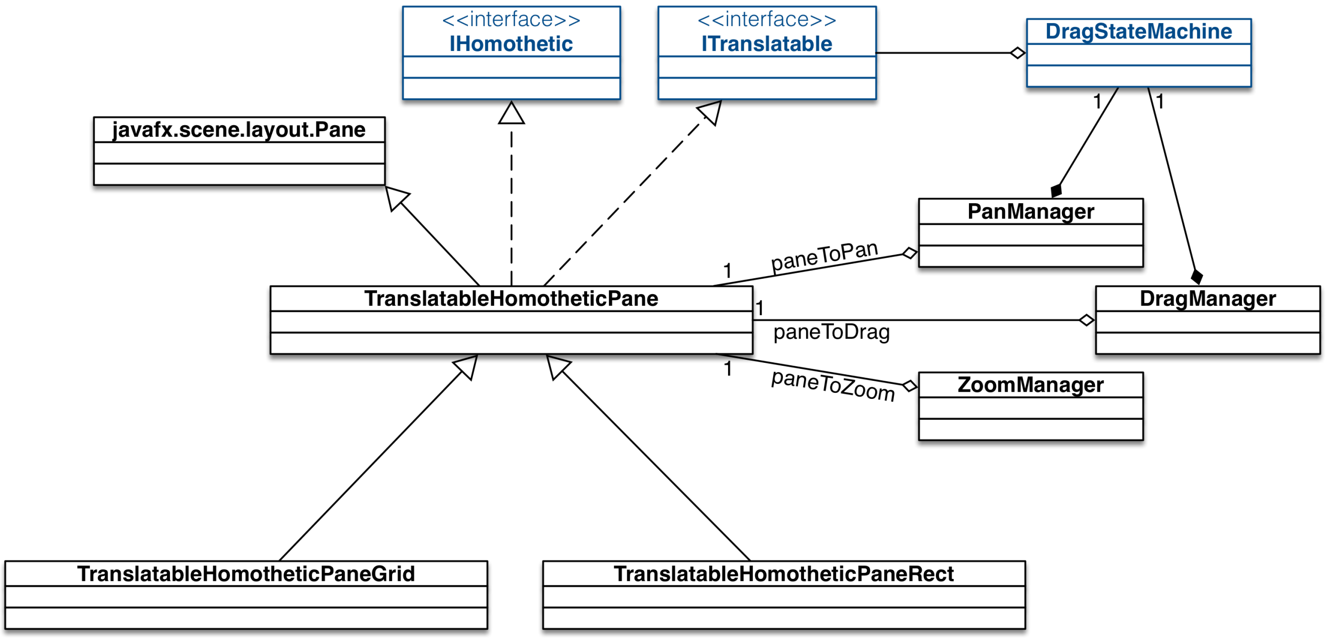
In *PW3* you have realized *pan* and *drag* interactions. But your code was fully coupled to *JavaFX*, which is unsatisfactory in terms of software architecture, since the drag logic is always the same. For reusability purposes it would be more satisfactory to create a graphics library agnostic logic mechanism and use it in conjunction with the library of your choice. To do so you transform your code and use a *FSM*.

1.1 Go back to the project created in *PW3* or create a new project and copy the whole code coming from the original project. Then add the provided code of the library `fr.liienac.statemachine` in your source folder. We provide the code rather than a *jar* for you to read and understand. It contains some interesting things that we don't have time to cover, like genericity.

Create a new package `javafxdragpanzoom.statemachines`.

In this package create a new class `DragStateMachine` extending the class `StateMachine`, and code the *FSM* we have drawn during the mini-course. Remember that in order to be agnostic, this class should not contain any *JavaFX* code. It will therefore be in charge of manipulating an `ITranslatable` thanks to its method `translate`, according to the new class diagram below, and will be sent (from `PanManager` and `DragManager`) only abstract events (use the ones defined in the package `event` of the library `statemachine`). This way, all the blue part of the diagram will be directly reusable with another graphics library.

Once your *FSM* is finished, modify your classes `PanManager` and `DragManager` to work with it.



1.2 Now create a new class `DragStateMachineHyst` and code the drag *FSM* with hysteresis. You will use it only for the *drag*.

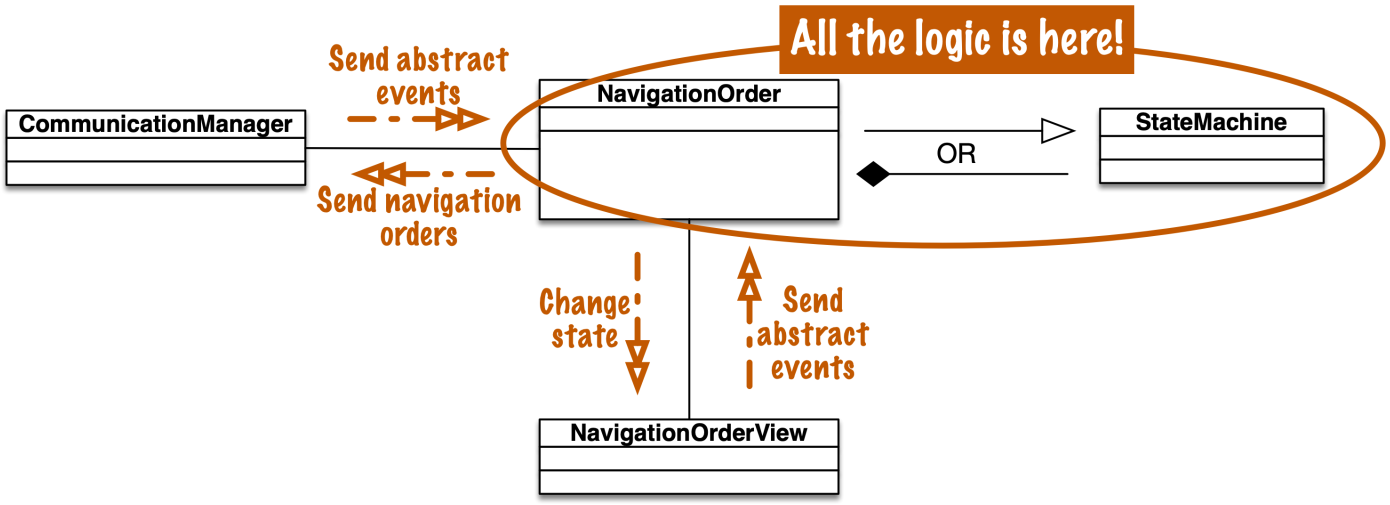
1.3 Is it necessary to use a *FSM* for the zoom interactions?

Exercise 2: *FSM* to manage the visual states of a graphical component (navigation order)

Let's now discuss how to use a *FSM* to manage the visual states of a graphical component which will be a *WIMP* widget allowing to visualize/send a navigation order (heading, direct, speed…). Let's first use a symbolic name like `NavigationOrder`. When you code, you will have to choose which order you want to implement.

One way to do things cleanly is to take inspiration from the TicTacToe game in *PW2*, and create a set of two associated classes that can be considered as a *view* and a *model*:

- a class `NavigationOrderView` (the *view*), which is dedicated to the visual representation of (and the interactions with) a given navigation order,
- and a class `NavigationOrder` (the *model*), which contains all the logic and is "controlled" by a *FSM*, according to the following diagram (informally combining class diagram syntax and free annotations).



2.1 Identify the different states a navigation order must have. To do so, think safety and usability first: each step of the process (preparing the order, sending it to the airborne system, waiting for an answer…) must be taken into account in order to improve the remote pilot's situational awareness by offering him a good feedback.

2.2 Now you have an idea of the necessary states, draw the associated *FSM* on paper. To do so:

- identify the user's actions on `NavigationOrderView` that will trigger state changes,
- identify the airborne messages that will trigger state changes,
- associate all of them to new abstract events (there will be new classes, just like the `Press`, `Move`… classes used in the exercise 1) and useful information to pass with these events,
- draw the states identified in the previous question,
- between the states, draw the transitions that will be triggered by the events,
- on each state and/or transition, specify the feedback to apply to `NavigationOrderView` (the `enter()` and/or the `action()` methods).

2.3 As you can see a state machine can quickly become complex when we take of a lot of things into account. But your code will always be more verifiable and more maintainable than it would be without the state machine. As a full implementation won't be possible during this work session, negotiate a minimal version to implement at first.

2.4 Code it:

- create the different event classes,
- choose a specific navigation order and create the associated class `XxxOrder` as a subclass of `StateMachine` (this will be simpler than the composition alternative, why?),
- populate it with your states and transitions.

2.5 Create a very simple `XxxOrderView` associated to your `XxxOrder` in order to test it:

- extend for example a `HBox` and just add buttons (as many as there are events coming from the view),
- add interactivity on these buttons (i.e. send the corresponding event with predetermined values to the *FSM*),
- in response to these events (and also the events coming from `CommunicationManager`) the *FSM* will activate transitions (i.e. do some state changes and perform actions on the view). In order to facilitate these actions create corresponding methods in the view, so that *FSM* will just have to invoke these methods,
- in these methods, just print a simple message to confirm the actions.

2.6 Create a fake `CommunicationManager` (no real communication) that will immediately answer predetermined events to the orders sent by your `XxxOrder` class.

2.7 Test your code. You can also use the "debug" mode of the class `StateMachine` (`setDebug(true)`) to have additional information printed in the standard output (transitions and state changes).

Now your super *FSM*-enabled navigation order is tested, you can turn your test widget into a more realistic view. Then you'll be able to start using it in your project with the real `CommunicationManager`, and make your navigation order widget evolve towards a more complete component that will take safety and usability considerations into account, provided that it is possible under this project's conditions, with the other components (*Rejeu*…). But this is your job to design the ideal model, to check it against your constraints, and to make justified assumptions/simplifications before developing your proof of concept.

Now these aspects have been discussed, the following questions will guide you in your choices to design a better software architecture, and therefore ease the evolutions to come…

2.8 As you should have noticed the *FSM* library is not *JavaFX*-enabled (there is no observability mechanism). So the view cannot observe the model and the notification of state changes is therefore not automatic. So the model must know the view in order to explicitly send the state changes notifications.

So, in order to work as specified, `NavigationOrder` must know both `NavigationOrderView` (its associated view) and `CommunicationManager` (its way to communicate with the airborne system). And reciprocally, both `NavigationOrderView` and `CommunicationManager` must know `NavigationOrder`.

Here again, remember the way TicTacToe game was architected in *PW2* and propose:

- a way to allow `NavigationOrder` to know its view without passing any reference in its constructor,
- a place to store `NavigationOrder`, with a convenient way to access it from the rest of the application (hints: we want to be sure that there is only one instance and we don't want to pass references everywhere…),
- a way to access the useful methods (yet to create) in `CommunicationManager` from `NavigationOrder`, in order to send the navigation order to the airborne system.

2.9 The first bullet of the previous question has another advantage: it will allow to easily create several views associated to the same model, which will be useful later when we add *direct manipulation* navigation order widgets to our first *WIMP* widgets and make them all work together.

In order to prepare the ground for this addition, propose a way to connect several different views (not just a `NavigationOrderView`) to the `NavigationOrder` model simultaneously. This will require a small modification in the model, as well as a factorization for potentially associated views.

2.10 The cherry on top: *Genericity*

As you've seen from the beginning of this exercise, most, if not all, navigation order widgets will look the same and will behave in the same way (they have the same *look & feel*). The only differences will be the title and the type of the value associated with the order (double, string...).

You could therefore:

- transform `NavigationOrder` into a generic class,
- transform in the same way `NavigationOrderView` into a generic base class factoring the main part of the widgets,
- and create several subclasses associated with the different navigation orders, specifying the type of the value they manipulate.

But it's not that simple: as a consequence of not knowing the type of the value, you will not know how to retrieve (parse) this value in the input field, nor the specific command(s) to send to `CommunicationManager`. So you will have to create generic intermediate classes (or interfaces) to handle this…

Genericity, if implemented, will be scored as a bonus in your project.

Now we've discussed everything it's time to send navigation orders!